Formation R

Benoît Lepage

2025-08-06

Contents

Chapter 1

Bienvenue sur cette formation au logiciel R

R est un logiciel accessible gratuitement permettant de réaliser des analyses statistiques dans un environnement windows, macOS ou Linux.

1.1 Pourquoi choisir R ?

Le logiciel est gratuit, très complet, avec une communauté d'utilisateurs très active dans le monde entier. Il est fréquent que les nouvelles méthodes d'analyses statistiques développées dans les équipes académiques soient d'abord mises à disposition sur R.

Le logiciel R repose sur l'utilisation de **scripts** dans lesquels nous allons **programmer** les analyses statistiques. Cette écriture sous forme de programmation peut paraître austère à première vue, mais est indispensable pour permettre la **reproductibilité** et la **transparence** des analyses. La même démarche de programmation est utilisée dans tous les logiciels statistiques professionnels (Stata, SAS, Python, Matlab, etc).

Pour utiliser R, les premières choses à faire sont de :

- télécharger le logiciel R
- et télécharger un Environnement de Développement Intégré (IDE) comme RStudio.

1.2 Téléchargez le logiciel R

Vous pouvez télécharger la dernière version stable du logiciel R sur le site du R project.

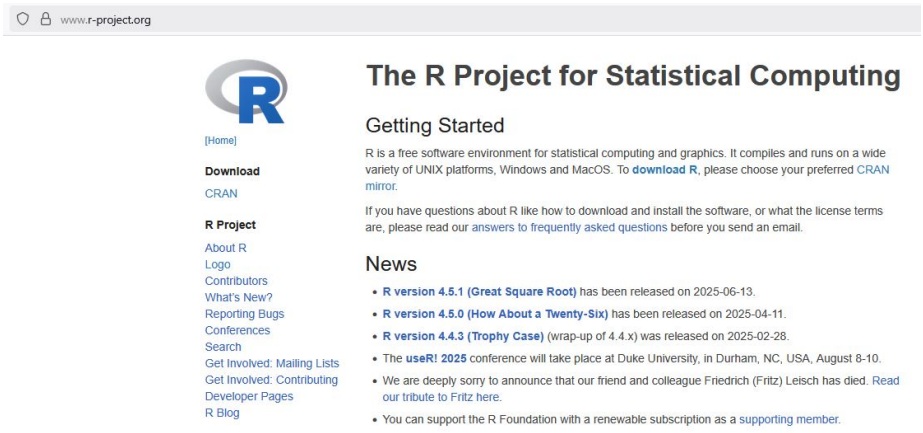

Figure 1.1: Site du R project, en juillet 2025

Cliquez sur “download R”, choisissez un site mirroir (par exemple un des sites en France).

Puis téléchargez la version de R en fonction de votre système d’exploitation (Windows, macOS ou Linux).

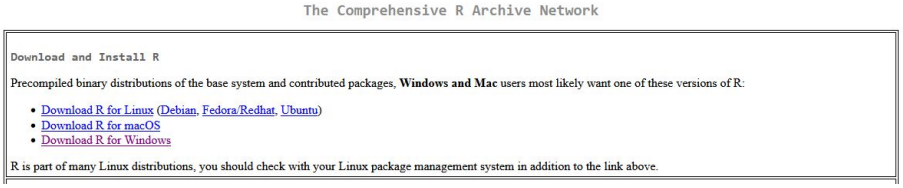

Figure 1.2: Choisissez la version adaptée à votre système d’exploitation

Enfin, installez R à partir du fichier d’installation que vous venez de télécharger.

1.2.1 Ouvrez le logiciel R

Si vous ouvrez le logiciel R, vous aller trouver l’interface graphique de R (*RGui* pour *R Graphical user interface*). Il est possible de faire vos analyses statistiques à partir de cette interface graphique, mais elle est très très austère.

Plutôt que d’utiliser cette interface RGui, nous vous recommandons fortement d’utiliser un Environnement de Développement Intégré (IDE), comme RStudio,

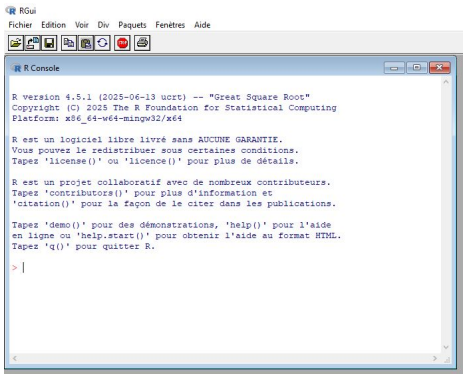

Figure 1.3: L'interface graphique de R (RGui)

qui vous facilitera grandement la vie pour utiliser un logiciel statistique qui repose sur de la programmation.

1.3 Téléchargez un IDE (RStudio recommandé)

RStudio est un environnement qui permet d'utiliser R, mais également d'autres logiciels de programmation comme Python, SQL, Stan, C++, etc. Cet environnement vous facilitera le travail pour :

- éditer vos scripts de programmation,
- accéder à la console,
- visualiser vos environnements de travail avec les fichiers et les objets qu'il contient,
- visualiser vos sorties graphiques et certaines tables d'analyses,
- visualiser vos données,
- visualiser les fichiers d'aide,
- gérer les *packages* permettant de faire des analyses spécifiques,
- et bien d'autres choses encore.

Par exemple, le tutoriel que vous êtes en train de lire a été créé à partir du package `bookdown` avec le logiciels R, au sein de l'IDE RStudio,

Vous pouvez télécharger la dernière version de RStudio sur le site de la compagnie Posit. Choisissez la version qui est adaptée à votre système d'exploitation (Windows, macOS ou Linux).

Puis, installez RStudio à partir du fichier d'installation que vous venez de télécharger.

Figure 1.4: téléchargez RStudio

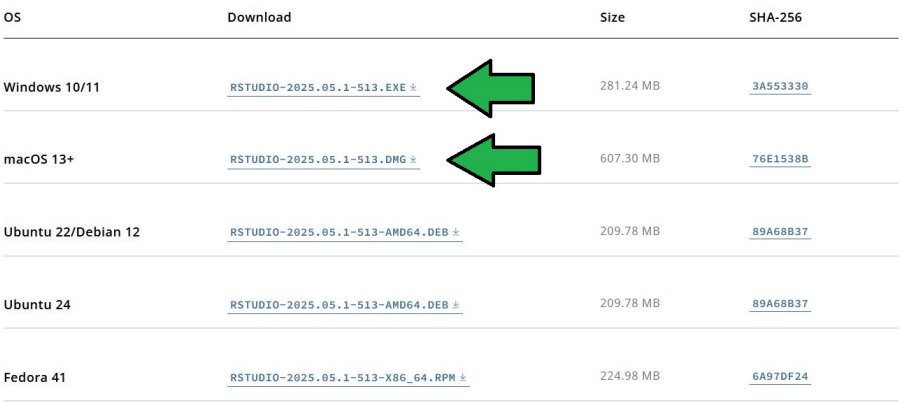

Figure 1.5: téléchargez RStudio

1.3.1 Ouvrez l'IDE RStudio

Ouvrez RStudio, puis commencez par ouvrir un **script**

- à partir du menu File > New File > R script
- ou bien en utilisant le raccourci Ctrl+Maj+N sur windows
- ou bien en cliquant sur le petit fichier blanc avec un + vert en haut à gauche, puis choisir "R script"

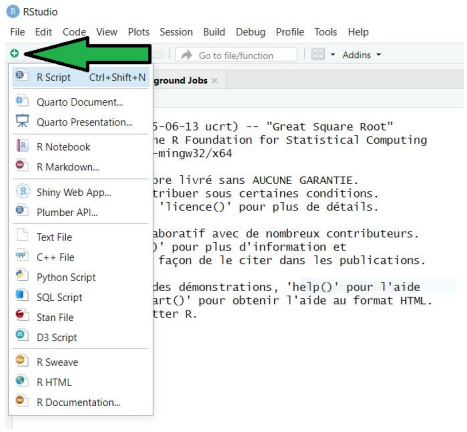

Figure 1.6: Ouvrir un nouveau script

L'interface de RStudio contient un menu, 4 quadrants et des sous-menus et boutons dans chaque cadrant.

Les menus qui vous seront le plus utiles sont :

- Dans le menu principal,
  - le menu *File* vous permettra de créer de nouveaux fichiers, d'ouvrir des fichiers déjà existants, de sauver vos fichiers, d'importer des bases de données, etc.
  - le menu *Tools > Install packages...* pour installer de nouveaux packages
  - le menu *Tools > Global Options...* vous permet de choisir la version du logiciel R à utiliser (onglet "R General") ou bien de changer l'aspect graphique de l'environnement RStudio (onglet "Appearance", puis choisissez un "Editor theme", avec différentes interfaces claires ou sombres)
- Au sein du **script** (cadrant 1)
  - le bouton "disquette" permet de sauvegarder votre script

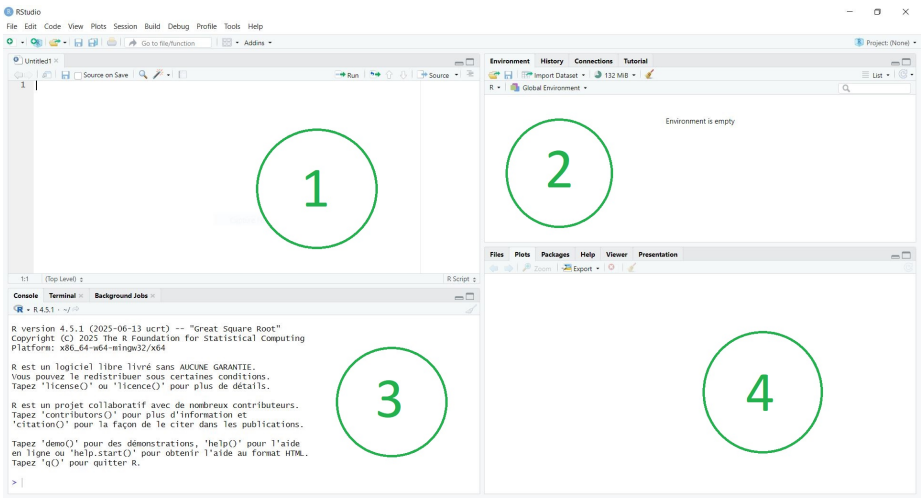

Figure 1.7: Les 4 cadrants de RStudio

  - le bouton “run” permet de faire tourner votre programme d’analyse (les lignes que vous avez sélectionnées). Par exemple, tapez la commande suivante dans le script, sélectionnez la ligne et cliquez sur le bouton “run” (ou avec un raccourci clavier `ctrl`+`entrée` sur windows, ou encore `command`+`entrée` sur macOS).

```
print("Hello Toulouse")
```

et vous devriez voir la commande `> print("Hello Toulouse")` puis son résultat `"Hello Toulouse"` dans l’onglet **console** du cadrant 3.

- Au sein du cadrant 3, l’onglet le plus utile pour pour les débutants est l’onglet **console**
  - la console est la même que la console affichée dans l’interface RGui du logiciel R que l’on a vu au paragraphe 1.2.1.
  - la console commence par afficher la version de R en cours d’utilisation
  - vous pouvez y saisir des commandes et obtenir directement leurs résultats, par exemple si vous tapez dans la console `4+9`, vous obtiendrez directement le résultat `13`. **Attention, les commandes que vous saisissez directement dans la console ne seront pas sauvegardées. Si vous voulez sauvegarder des commandes, il faut utiliser le *script* (cadrant 1)**

```
4+9
```

```
## [1] 13
```

- Au sein du cadrant 2, l'onglet le plus utile pour les débutants est l'onglet **Environment**
  - cet onglet vous permettra de visualiser les "objets R" créés pendant vos analyses.
  - Par exemple si vous saisissez `v <- 1:10` dans la console, vous allez voir apparaître l'objet `v` dans l'environnement de travail (il s'agit d'un vecteur de 1 à 10, nommé "v").
- Au sein du cadrant 4, les onglets les plus utiles pour les débutants sont :
  - l'onglet "File" qui contient les dossiers et fichiers au sein d'un dossier de travail (voir le chapitre 3 pour créer et organiser un dossier de travail associé à un "projet R")
  - l'onglet "Plots" où vous retrouverez vos sorties graphiques. Au sein de cet onglet, vous trouverez un menu pour exporter vos graphiques selon différents formats. Des boutons permettent également de zoomer et d'effacer les graphiques. Par exemple, si vous saisissez `hist(rnorm(10000))` dans la console, un histogramme d'une distribution normale centrée réduite va apparaître. Vous pouvez effacer la figure en cliquant sur le bouton avec la croix rouge (efface la figure actuelle) ou le balet (efface l'ensemble des figures).
  - l'onglet "Packages" où vous pourrez activer, désactiver ou mettre à jour les packages qui ont été téléchargés.
  - l'onglet "Help" où vous trouverez de l'aide. Par exemple si vous saisissez `help(mean)` dans la console, l'aide de la commande `mean` va s'afficher. Vous pouvez également utiliser le champ de recherche de fonctions dans le menu "Help".

1.4 Trouver de l'aide sur R

De nombreuses ressources sont disponibles pour vous aider à utiliser R :

- Les pages d'aide en ligne de R, qui apparaîssent directement dans RStudio. Vous pouvez obtenir de l'aide sur des fonctions et des packages :
  - en appliquant une recherche par mot clé dans le champ de recherche de l'onglet "help"
  - en utilisant directement dans la console la fonction `help.search()` ou `??` associée à un mot clé (par exemple `help.search(student)` ou `??student`), ou la fonction `help()` associée à une fonction (par exemple `help(t.test)` ou `?t.test`)

  - quand vous rencontrerez une nouvelle fonction (dans ce cours ou bien dans votre pratique), nous vous conseillons d'aller systématiquement regarder l'aide de cette fonction pour bien comprendre comment l'utiliser.
  - Ces pages d'aide suivent la structure suivante :
    * une partie "Description" qui décrit en quelques phrases ce que fait la fonction
    * une partie "Usage" qui décrit la syntaxe de la fonction avec ses arguments
    * une partie "Argument" qui précise comment renseigner les arguments de la fonction
    * une partie "Detail" qui décrit en détail comment utiliser la fonction et ses arguments
    * une partie "Value" qui décrit les sorties (les résultats) de la fonction, avec les éventuels sous-objets de la sortie
    * une partie "Exemples" qui indique quelques exemple que vous pouvez directement lancer en cliquant sur "Run examples"

- Des fiches "mémoires" *cheat sheets* qui résument les principales fonctions :

  - pour les commandes R bases
  - des bonnes pratiques sur R
  - méthodes de visualisation avec le package ggplot2
  - méthodes de manipulation de données avec le package tidyr
  - méthodes de transformation de données avec le package dplyr
  - l'utilisation du package data.table
  - l'utilisation du package stringr pour manipuler les chaînes de caractères, avec la fiche stringr
  - la manipulation de dates Dates and times with lubridate et la fiche lubridate

- De nombreux livres et tutoriels disponibles gratuitement en ligne :

  - Le guide R de Joseph Larmarange, est un guide très complet et didactique, en français
  - L'Epidemiologist R Handbook est un tutoriel en anglais pour l'utilisation de R par des épidémiologistes
  - Software carpentry met à disposition des guides introductifs bien réalisés en anglais, par exemple R for Reproducible Scientific Analysis ou encore Programming with R
  - le livre R for Data Science propose une introduction très complète pour l'analyse de données descriptive principalement basée sur la suite de packages du Tidyverse
  - le package R swirl propose une formation interactive directement dans la console de RStudio, téléchargez le package

(`install.packages("swirl")`), chargez le package (`library("swirl")`) et laissez vous guider après avoir saisi `swirl()` dans la console.
  - RStudio Education liste plusieurs ressources intéressantes pour les débutants
  - Le site CRAN a des manuels assez complets, par exemple la page R Language Definition est une introduction assez complète au langage de programmation R. La page CRAN Task Views liste les packages qui sont disponibles par thématique ou type d'anlayse.

- Rechercher à l'aide d'un moteur de recherche (google, DuckDuckGo, Bing, etc). Ces recherches vous amèneront régulièrement vers des forums de discussion comme stackoverflow ou stackExchange. Si vous rencontrez une erreur ou une difficulté, il y a toutes les chances que d'autres personnes aient déjà rencontré ces erreurs et difficultés avant vous, et que des solutions détaillées soient proposées dans ces forums.

- Les chatbots de type GPT, Copilot ou Gemini : dans le cadre de votre formation au logiciel R, nous vous déconseillons l'utilisation de ces outils basés sur des LLM. Ces outils posent de nombreux problèmes en termes de transparence, de respect de droit d'auteur, d'impact environnemental, de déqualification (délegation de compétences pour rechercher de l'information, perte d'esprit critique), de dépendance aux Gafams, de dégradation des systèmes d'information, etc. Par ailleurs, bien qu'ils peuvent apporter des solutions fonctionnelles, il est bien plus utile d'avoir une bonne compréhension des bases de programmations sur R avant d'utiliser de tels outils : sans une bonne compréhension de la logique de programmation et des principales fonctions de R, vous aurez des difficultés à évaluer la fiabilité des solutions proposées, à vous débloquez en cas de problèmes, ou encore à adapter vos prompts pour obtenir de meilleures réponses. Les ressources décrites précédemment devraient vous permettre d'apporter efficacement des réponses à vos questions.

1.5 Conventions d'écriture

Certaines conventions ont été proposées pour faciliter l'écriture et la lecture du code de programmation dans R. Elles ne sont pas obligatoires, mais nous vous encourrageons à les suivre.

Par exemple, le *Tidyverse style guide* :

- nommer les variables et les fonctions en lettres minuscules, à l'aide de mots et de chiffres séparés par `_` (*underscore*). Par exemple `csp_1`. Cette convention fait référence au style "snake case"
- ajouter un espace après une virgule, par exemple `x[2, 5]`

- ajouter des espaces avant et après les opérateurs arithmétiques, par exemple `x <- (1 + 2) / 5`, à quelques exceptions près (pas d'espace avant ou après le signe "puissance" `^`) `y <- x^2 + 3`

Chapter 2

Les objets dans R

La programmation R repose sur des *objets*, qui apparaîtront dans la fenêtre *Environment* de RStudio.

Les objets les plus élémentaires dans R sont des **vecteurs**. Un vecteur contient une série de valeurs (des nombres, des chaînes de caractères, ou des données plus complexes). Il y a deux types de vecteurs :

- les **vecteurs atomiques** (les valeurs d'un vecteur atomique doivent toujours être du même type),
- les **listes** (les valeurs d'une liste peuvent être de différents types).

2.1 Manipuler les objets dans l'environnement

Voici quelques commandes de gestion des objets dans votre environnement :

- dans la console, commencez par créer les objets suivants. Pour **assigner une ou plusieurs valeurs à un objet**, on utilise une flèche dirigée vers la gauche `<-`. Vous verrez apparaître ces objets dans la fenêtre *Environment*.

```
# note : le signe dièze (#) permet d'ajouter des commentaires dans le code
# - le 1er objet est un vecteur de 10 nombres entiers de 1 à 10
# - le 2ème objet est un vecteur de 3 lettres A, B et C
# - le 3ème objet est un vecteur de 2 réels, calculés par 2 opérations
# - le 4ème objet est une fonction qui ajoute 2 aux éléments du vecteur x
# - le 5ème objet est un scalaire égal à 42
objet_1 <- c(1:10)
objet_2 <- c("A", "B", "C")
```

```
objet_3 <- c(10 / 3, 4 * 5)
objet_4 <- function(x) {x + 2}
objet_5 <- 42
```

- la commande `ls()` permet de **lister** les objets dans l'environnement.
- la commande `rm()` permet de **supprimer** (*remove*) un ou plusieurs objets de l'environnement.

```
ls()
```

```
## [1] "objet_1" "objet_2" "objet_3" "objet_4" "objet_5"
```

```
rm(objet_2, objet_5)
rm(list = ls()) # pour supprimer tous les objets présents dans l'environnement
```

Conseils pour nommer un objet dans R :

- Utiliser des noms courts, mais lisibles, faciles à comprendre
- Les noms doivent commencer par une lettre (pas par un nombre) et ne peuvent pas contenir d'espace. A noter qu'ils peuvent contenir des points, comme par exemple `nom.variable`.
- Evitez d'utiliser des noms de variables et de fonctions déjà existants dans R (par exemple `mean` qui risque de porter à confusion avec la fonction `mean()` : il vaut mieux utiliser `mean_variable`)
- Ecrire en minuscule avec underscore pour séparer les mots (par exemple `date_naissance`)

2.2 Principaux types de données

Les données peuvent être de différents types. Les 4 principaux types sont :

- les **nombres réels** (`?double`), par exemple `12.43`.
- les **nombres entiers** (`?integer`). Les nombres entiers sont saisis en ajoutant `L` à droite du nombre, par exemple `5L`.
- les **chaînes de caractères textuels** (`?character`), définis avec des guillemets simples ou doubles , par exemple `'bonjour'` ou `"au revoir"`
- les **valeurs logiques** (`?logical`), avec deux valeurs possibles :
  - valeur booléenne *vraie*, notée `TRUE` ou bien `T`
  - valeur booléenne *fausse*, notée `FALSE` ou bien `F`

On peut également trouver des types de données un peu plus sophistiquées, construites à partir des principaux types :

- des **variables qualitatives** qui peuvent être **nominales** (`?factor`) ou **ordinales** (`?ordered`). Ces variables sont construites sur des nombres entiers.
- des **dates** (`?Date`), qui sont construites sur des nombres réels.
- des **dates-heure** (`?POSIXct`), qui sont construites sur des nombres réels.
- des **durées** (`?difftime`), construites sur des nombres réels.

On a également un vecteur particulier qui est le **vecteur nul** et se note `NULL`. Le vecteur nul a une longueur de 0 et ne peut avoir aucun attribut (les notions de longueur et d'attribut d'un vecteur seront vues plus bas).

2.2.1 Données manquantes

Quel que soit le type de données, les données manquantes se notent `NA` (not applicable).

Attention à ne pas confondre le vecteur nul `NULL` et les données manquante `NA`.

2.2.2 Décrire le type de l'objet ♠

Les paragraphes avec un ♠ présentent des notions plus avancées, si vous êtes en phase d'apprentissage, vous pouvez aller directement au paragraphe suivant.

On peut décrire quel est le type de l'objet avec les fonctions `typeof` (le type le plus élémentaire), `mode` et `storage.mode` (mode de l'objet et mode de stockage de l'objet selon un regroupement un peu plus large).

Par exemple, les nombres réels (`double`) et les nombres entiers (`integer`) sont du mode `numeric`.

```
# les valeurs réelles ('double') et les entiers ('integer') sont de mode 'numeric'
typeof(2.53) # un réel
typeof(5L) # et un entier
mode(2.53) # sont de mode 'numeric'
mode(5L)
storage.mode(2.53)
storage.mode(5L)

# les chaînes de caractères sont de type et de mode 'character'
typeof(c("hello", "Toulouse"))
mode(c("hello", "Toulouse"))
```

```r
# les valeurs logiques sont de type 'logical'
typeof(c(TRUE, FALSE, FALSE))
mode(c(TRUE, FALSE, FALSE))
```

x	typeof(x)	mode(x)	storage.mode(x)
2.53	"double"	"numeric"	"double"
5L	"integer"	"numeric"	"integer"
"bonjour"	"character"	"character"	"character"
TRUE	"logical"	"logical"	"logical"
as.Date("2025-07-01")	"double"	"numeric"	"double"

Les fonctions `as.double`, `as.integer`, `as.character`, `as.logical` permettent de forcer par coercition le type d'un objet *en tant que* réel, entier, chaîne de caractères, logique.

```r
as.double(5L) # définit un nombre entier en tant que nombre réel
as.integer(4.95) # définit un réel en tant qu'entier, seul l'entier est conservé
as.character(4.95) # définit un nombre en tant que chaîne de caractères

# définir une valeur logique TRUE et FALSE en tant que valeur numérique
# ou en tant qu'entier donne les valeurs 1 et 0, respectivement
as.numeric(TRUE)
as.numeric(FALSE)

# définir le nombre 0 en tant que valeur logique donne la valeur FALSE
as.logical(0)

# définir tout nombre différent de 0 en tant que valeur logique
# donne la valeur TRUE
as.logical(-14)
as.logical(1)
as.logical(4.95)
```

Les fonctions `is.double`, `is.integer`, `is.character`, `is.logical` permettent d'évaluer si un objet est de type réel, entier, textuel, logique.

```r
is.double(5L) # FALSE, un nombre de type "entier" n'est pas de type "réel"
              # attention, en math, les nombres entiers font partie des réels !
is.integer(4.95) # FALSE, un nombre de type "réel" n'est pas de type "entier"
is.numeric("bonjour") # FALSE "bonjour" est une chaîne de caractères
is.character("bonjour") # TRUE, "bonjour" est bien une chaîne de caractères
is.character(4.95) # FALSE, 4.95 est un objet numérique
```

```r
is.logical(1) # FALSE, 1 est un objet numérique
is.logical(as.logical(1)) # TRUE, as.logical(1) = TRUE, qui est un objet logique
is.logical(TRUE) # TRUE est bien un objet logique
```

2.3 Principales structures de données

Les principales structures de données que nous allons détailler dans la suite de ce chapitre sont :

- les **vecteurs atomiques** (`?c()`), qui doivent toujours comporter des valeurs du même type. Les vecteurs qui ne comportent qu'une seule valeur sont appelés des "scalaires" ;
- les **listes** (`?list`), qui sont également des vecteurs, mais peuvent comporter des valeurs de types différents ;
- les **matrices** (`?matrix`, `?array`), qui sont des vecteurs atomiques réarrangés sous forme de tables à 2 dimensions ou plus ;
- les **bases de données** (`?data.frames`). Les bases de données sont des listes de vecteurs atomiques de même longueur. Il existe d'autres formats de base de données qui seront présentés plus tard (avec les packages `tidyverse` et `data.table`).

2.4 Objet à une seule valeur (scalaire ou texte)

2.4.1 Scalaires

Assignez les valeurs 4 et 5 à deux objets

```r
x_1 <- 4
x_2 <- 5
```

2.4.2 Opérations mathématiques sur les scalaires

2.4.2.1 Calculatrice

On peut utiliser les opérations classiques, comme sur une calculatrice :

- `+` pour **additionner**
- `-` pour **soustraire**
- `*` pour **multiplier**
- `/` pour **diviser**

- `^` pour mettre à la **puissance**
- `e` pour la **notation scientifique**

```
x_1 + x_2 # 4 + 5 = 9
10 - x_1 # 10 - 4 = 6
x_1 * x_2 #  4 * 5 = 20
20 / x_2 # 20 / 5 = 4
x_1^2 #  4^2 = 16
10^-1 # 1/10 = 0.1
25^(0.5) # racine carrée de 25 (puissance 1/2)

# notation scientifique pour les grands et petits nombres
1/1000000 # 1 pour 1 million = 1e-6
1/1e6
1e6 * 1000 # 1 million * 1000 = 1 milliard
```

2.4.2.2 Fonctions mathématiques

Plusieurs fonctions mathématiques de bases sont implémentées nativement dans R :

- `log(x)` ou `log(x, base = exp(1))` pour le **logarithme** népérien,
- `log10(x)` pour le logarithme base 10, `log2(x)` pour le logarithme base 2,
- `log(x, base = b)` pour le logarithme base `b`,
- `exp(x)` pour l'**exponentielle** de `x`
- `sqrt(x)` pour la **racine carrée** de `x`
- `abs(x)` pour la **valeur absolue** de `x`
- les **fonctions trigonométriques** sont implémentées, avec `cos(x)`, `sin(x)`, `tan(x)` (cf. `?Trig`)
- la **constante** π est implémentée avec `pi` (cf. `?Constants`)

Si vous appliquer une fonction à une valeur qui ne fait pas partie du domaine de définition de la fonction, le résultat sera une valeur manquante notée `NaN` (*not a number*). Un message d'avertissement va apparaître si vous appliquez une fonction en dehors de son domaine de définition.

Les notions de + l'infini et - l'infini sont notées `Inf` et `-Inf`.

```
# logarithmes et exponentielles
log(1)
log10(100)
log(100, base = 10)
exp(1)

# racine carrée
```

```r
sqrt(x_2^2)

# valeur absolue
abs(10)
abs(-10)

# fonctions trigonométriques
cos(1)
sin(1)
tan(1)
pi
2 * pi * 10 # circonférence d'un cercle de rayon 10

# si on utilise une valeur en dehors du domaine d'application de la fonction
log(-1) # NaN, car -1 est en dehors du domaine de définition de la fonction log
sqrt(-2) # -2 est en dehors du domaine de définition de la fonction racine carrée

# notions de + ou - l'infini
1 / 0 # [1] Inf
-1 / 0 # [1] -Inf
```

2.4.2.3 Fonctions d'arrondi ♠

Plusieurs fonctions sont disponibles dans R pour arrondir une valeur (cf. `?Round`) :

- la fonction `round()` est utile pour arrondir les décimales. Il faut préciser en argument, le nombre de chiffres après la virgule. **Attention** : si le nombre se termine par un 5, l'arrondi se fait vers le chiffre pair le plus proche : 4.45 s'arrondit à 4.4 et 4.75 s'arrondit à 4.8
- la fonction `signif()` arrondit aux chiffres les plus significatifs (les plus grands)
- la fonction `floor()` arrondit la valeur à l'entier inférieur
- la fonction `ceiling()` arrondit la valeur à l'entier supérieur
- la fonction `trunc()` ne garde que les entiers, sans arrondir

```r
## fonction round()
# l'argument digits permet de définir le nombre de chiffres après la virgule
# exemple si vous voulez arrondir à 2 chiffres après la virgule
round(0.09400, digits = 2) # 0.09
round(0.08600, digits = 2) # 0.09
# arroudir à 1 chiffre après la virgule
round(4.450, digits = 1) # 4.4 ; arrondit au chiffre pari plus proche
round(4.750, digits = 1) # 4.8 ; arrondit au chiffre pari la plus proche
```

```r
# pour arrondir une valeur 5, le résultat va vers le chiffre pair le plus proche
round(4.5, digits = 0) # 4 ; arrondit au chiffre pair le plus proche
round(1.5, digits = 0) # 2 ; arrondit au chiffre pair le plus proche

## fonction signif()
# on garde les valeurs les plus significative, définie par l'argument digits
signif(123.456789, digits = 1) # 100
signif(123.456789, digits = 2) # 120
signif(123.456789, digits = 3) # 123
signif(123.456789, digits = 4) # 123.5
signif(123.456789, digits = 5) # 123.46 arrondit au chiffre pair le plus proche
signif(4.45, digits = 3) # 4.45
signif(4.45, digits = 2) # 4.4 ; arrondit au chiffre pair le plus proche
signif(4.75, digits = 3) # 4.75
signif(4.75, digits = 2) # 4.8 ; arrondit au chiffre pair le plus proche

## fonction trunc() supprime simplement les décimales
# note : ici, il n'y a pas d'arrondi vers la chiffre pair la plus proche)
trunc(123.456) # 123
trunc(4.5) # 4
trunc(1.5) # 1

## la fonction floor() arrondit à l'entier inférieur
floor(4.1) # 4
floor(4.9) # 4

## la fonction ceiling() arrondit à l'entier supérieur
ceiling(4.1) # 5
ceiling(4.9) # 5
```

2.4.3 Concaténation de chaînes de caractères

On peut concatener deux objets en chaînes de caractères :

- la fonction `paste()` concatène les chaînes de caractères en séparant les vecteurs par un espace (argument par défaut, cf `?paste`). Cet argument peut être modifié.
- la fonction `paste0()` concatène les chaînes de caractères sans espace.

```r
x1 <- "Bonjour"
x2 <- "Toulouse"
paste(x1, x2)
```

```
## [1] "Bonjour Toulouse"
```

```r
paste0(x1, x2)
```

```
## [1] "BonjourToulouse"
```

```r
paste(x1, x2, sep = ", ") # ici on sépare x1 et x2 par une virgule et un espace
```

```
## [1] "Bonjour, Toulouse"
```

```r
# vous pouvez inclure des nombres qui seront transformés en caractères
paste0(x1, 123, x2)
```

```
## [1] "Bonjour123Toulouse"
```

2.4.4 Valeurs logiques `TRUE` et `FALSE`

2.4.4.1 Evaluer des conditions

Nous pouvons utiliser les opérateurs de comparaison ci-dessous pour évaluer des conditions :

- `==` ... est égal à ...
- `!=` ... est différent de ...
- `<` ... est inférieur à ...
- `>` ... est supérieur à ...
- `<=` ... est inférieur ou égal à ...
- `>=` ... est supérieur ou égal à ...
- `%in%` ... est inclus dans ...

Par exemple, nous pouvons évaluer les comparaisons suivantes, la réponse attendue est vraie (`TRUE`) ou fausse (`FALSE`).

```r
5 == 10 # est-ce que 5 est égal à 10 ?
5 != 10 # est-ce que 5 est différent de 10 ?
5 < 10 # est-ce que 5 est inférieur à 10 ?
5 > 10 # est-ce que 5 est supérieur à 10 ?
5 <= 5 # est-ce que 5 est inférieur ou égal à 5 ?
5 >= 5 # est-ce que 5 est supérieur ou égal à 5 ?
5 %in% c(4,5,6) # est-ce que 5 est inclus dans le vecteur (4,5,6) ?
5 %in% c(7,8,9) # est-ce que 5 est inclus dans le vecteur (7,8,9) ?
```

La fonction `identical` permet d'évaluer si deux objets sont exactement égaux. Elle peut s'appliquer à des valeurs simples mais aussi à des objets de plus grandes dimensions (vecteurs, matrices, bases de données, ...)

```
identical(5, 10) # équivalent à la commande 5 == 10
identical(c(1,2,3), c(1,2,3)) # les deux vecteurs (1,2,3) sont bien les mêmes
```

Une comparaison à une valeur manquante (`NA`) retournera une valeur manquante.

Attention, si vous souhaitez évaluer si une valeur est manquante, il faut utiliser la fonction `is.na(x)` (plutôt que `x == NA` qui est déconseillé).

```
NA < 10
is.na(10) # éviter d'utiliser 10 == NA pour tester si une valeur est manquante
is.na(NA)
is.na(c(1,2,3,NA,5,6,NA,8,9,10))
```

2.4.5 Opérations sur des valeurs logiques

On peut combiner des valeurs logiques avec les opérateurs logiques ET, OU, et NON (négation logique)

- `&` opérateur ET
- `|` opérateur OU (sur windows, combinaison de touches altgr + 6 ; sur macOS, combinaison de touche alt + maj + L)
- `!` opérateur NON (négation logique : “n’est pas”)

Les résultats attendus d’une combinaison d’opérateurs logiques sont résumés dans les table de vérité ci-dessous.

- Opérateur ET

a	b	a ET b
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

- Opérateur OU

a	b	a OU b
TRUE	TRUE	TRUE

a	b	a OU b
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

- Opérateur NON

a	NON a
TRUE	FALSE
FALSE	TRUE

```
# opérateur ET
TRUE & TRUE
TRUE & FALSE
FALSE & TRUE
FALSE & FALSE
(5 > 10) & (2 != 5) # TRUE ET TRUE donne TRUE
(5 > 10) & (2 == 5) # TRUE ET FALSE donne FALSE
(5 < 10) & (2 == 5) # FALSE ET FALSE donne FALSE

# opérateur OU
TRUE | TRUE
TRUE | FALSE
FALSE | TRUE
FALSE | FALSE
(5 < 10) | (2 != 5) # TRUE OU TRUE donne TRUE
(5 > 10) | (2 != 5) # FALSE OU TRUE donne TRUE
(5 > 10) | (2 == 5) # FALSE OU FALSE donne FALSE

# opérateur NON
!TRUE
!FALSE
!(5 < 10) # non-TRUE donne FALSE
!(5 > 10) # non-FALSE donne TRUE
```

Il existe également un opérateur `xor()` correspondant au OU EXCLUSIF (mais il semble peu utilisé en pratique) :

- Opérateur OU EXCLUSIF

a	b	a OU EXCLUSIF b
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

```
# opérateur OU EXCLUSIF
xor(TRUE, TRUE)
xor(TRUE, FALSE)
xor(FALSE, TRUE)
xor(FALSE, FALSE)
xor((5 < 10), (2 != 5)) # (TRUE) OU exclusif (TRUE) donne FALSE
xor((5 > 10), (2 != 5)) # (FALSE) OU exclusif (TRUE) donne TRUE
xor((2 != 5), (5 > 10)) # (TRUE) OU exclusif (FALSE) donne TRUE
xor((5 > 10), (2 == 5)) # (FALSE) OU exclusif (FALSE) donne FALSE
```

2.5 Les vecteurs atomiques c()

2.5.1 Création d'un vecteur atomique

La fonction c() (pour concaténer) permet de créer des vecteurs atomiques. Les valeurs d'un vecteur atomique doivent toutes être du même type et séparées par une virgule.

```
## Un vecteur de 3 nombres réels :
vect_dbl <- c(1, pi, 3.54)
print(vect_dbl) # [1] 1.000000 3.141593 3.540000

## Un vecteur de 5 entiers :
vect_int <- c(5L, 4L, -3L, 15L, -8L)
print(vect_int) # [1]  5  4 -3 15 -8

## Un vecteur de 4 chaînes de caractères :
vect_char <- c("a", "B", "XYZ", "HELLO")
vect_char # [1] "a"     "B"     "XYZ"   "HELLO"

## Un vecteur logique :
## pour les valeurs logiques, on peut utiliser les abbréviations T pour TRUE,
## et F pour FALSE
vect_logic <- c(TRUE, FALSE, FALSE, F, T, T)
print(vect_logic) # [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

Vous pouvez également combiner plusieurs vecteurs pour créer un vecteur plus long en indiquant plusieurs vecteurs au sein de la fonction `c()`.

```
vect1 <- c(1, 2, 3)
vect2 <- c(4, 5, 6)
vect3 <- c(7, 8, 9)
vect_combine <- c(vect1, vect2, vect3)
vect_combine
# [1] 1 2 3 4 5 6 7 8 9
```

2.5.2 Créer une séquence de valeurs

Pour créer une séquence de nombre entiers, ascendante ou descendante de 1 en 1, vous pouvez utiliser la commande :

```
vect_1a10 <- 1:10 # vecteur ascendant
vect_1a10 # [1]  1  2  3  4  5  6  7  8  9 10

vect_3_a_moins5 <- 3:-5 # vecteur descendant
vect_3_a_moins5 # [1] 3  2  1  0 -1 -2 -3 -4 -5
```

La commande `seq()` permet de définir des séquences de manière plus flexible, grâce aux arguments `from`, `to`, `by`, `length.out`, `along.with` (regardez l'aide `?seq()`).

```
seq(from = 1, to = 10) # vecteur de 1 à 10, équivalent à la commande 1:10
seq(from = 0, to = 10, by = 2) # vecteur de 0 à 10, de 2 en 2

seq(from = 1, to = 50, length.out = 5) # vecteur de 1 à 50, de longueur 5
# les intervalles entre les valeurs sont calculés automatiquement par la formule
# (valeur de départ - valeur d'arrivée) / (longueur totale - 1)

seq(from = 0, to = 60, along.with = c("dix", "vingt", "trente"))
# vecteur de 0 à 60, dont la longueur est égale à la longeur du vecteur
# c("dix","vingt","trente") (sa longeur est de 3)

# avec l'argument 'by', si le cycle ne tombe pas juste, la séquence s'arrête
# avant la dernière valeur indiquée par 'to = '
seq(from = 1, to = 10, by = 2)
# [1] 1 3 5 7 9
# la séquence ne va pas jusqu'à 10, elle s'arrête à 9, car 9 + 2 = 11
# ce qui dépasserait la valeur maximale demandée par 'to = 10'
```

On peut combiner des vecteurs de chaînes de caractères à des vecteurs numériques :

```r
paste0("L", seq(from = 1, to = 10))
# on obtient le vecteur :
# "L1"  "L2"  "L3"  "L4"  "L5"  "L6"  "L7"  "L8"  "L9"  "L10"
# dans cet exemple, le vecteur à une seule valeur c("L") est "recyclé" pour être
# concaténé à chacun des éléments du vecteur c(1,2,3,4,5,6,7,8,9,10)

paste0(c("A", "B", "C"), seq(from = 1, to = 10))
# on obtient le vecteur :
# "A1"  "B2"  "C3"  "A4"  "B5"  "C6"  "A7"  "B8"  "C9"  "A10"
# ici, le vecteur c("A", "B", "C") est "recyclé" pour être concaténé à chaque
# élément du vecteur seq(1,10)
```

2.5.3 Créer un vecteur de valeurs répétées

La fonction `rep` permet de créer des vecteurs de valeurs répétées. Les arguments `times`, `each`, `length.out` permettent de préciser comment les valeurs doivent être répétées (voir dans l'aide `?rep`)

```r
rep(5, times = 3) # répète la valeur 5, 3 fois.
# [1] 5 5 5

rep(1:4, times = 3) # repète 3 fois le vecteur c(1, 2, 3, 4)
# [1] 1 2 3 4 1 2 3 4 1 2 3 4

rep(1:4, each = 3) # chaque élément du vecteur c(1, 2, 3, 4) est répété 3 fois
# [1] 1 1 1 2 2 2 3 3 3 4 4 4

rep(1:4, length.out = 10) # le vecteur c(1,2,3,4) est répété dans un vecteur
                          # dont la longueur totale est de 10
# [1] 1 2 3 4 1 2 3 4 1 2
```

2.5.4 Opérations arithmétiques sur un vecteur

On appliquer une opération arithmétique avec un scalaire (une valeur simple) à chacun des éléments d'un vecteur :

```r
## additionne +2 à chaque élément de la séquence c(1,2,3,4,5)
seq(from = 1, to = 5) + 2 # cela donne 3 4 5 6 7

## soustrait -2 à chaque élément de la séquence c(1,2,3,4,5)
seq(from = 1, to = 5) - 2 # cela donne -1  0  1  2  3

## multiplie par à chaque élément de la séquence c(1,2,3,4,5)
```

```r
seq(from = 1, to = 5) * 5 # cela donne 5 10 15 20 25

## divise par 5 chaque élément de la séquence c(1,2,3,4,5)
seq(from = 1, to = 5) / 5 # cela donne 0.2 0.4 0.6 0.8 1.0
```

Si vous appliquez une opération arithmétique entre deux vecteurs de même longueur, l'opération se fera entre les 1ers éléments de chaque vecteur, puis entre les 2èmes éléments de chaque vecteur, etc.

```r
vec_A <- c(1, 2, 3, 4, 5)
vec_B <- c(-1, +1, -2, +2, -5)

# addition : c((1 + (-1)), (2 + 1), (3 + (-2)), (4 + 2), (5 + (-5)))
vec_A + vec_B # 0 3 1 6 0

# soustraction : c((1 - (-1)), (2 - 1), (3 - (-2)), (4 - 2), (5 - (-5)))
vec_A - vec_B # 2  1  5  2 10

# multiplication : c((1 * (-1)), (2 * 1), (3 * (-2)), (4 * 2), (5 * (-5)))
vec_A * vec_B  # -1   2  -6   8 -25

# division : c((1 / (-1)), (2 / 1), (3 / (-2)), (4 / 2), (5 / (-5)))
vec_A / vec_B # -1.0  2.0 -1.5  2.0 -1.0
```

Si vous appliquez une opération arithmétique entre deux vecteurs de longueur différente, l'opération se fera entre les 1ers éléments de chaque vecteur, puis entre les 2èmes éléments de chaque vecteur, etc. Lorsqu'on arrive au bout du vecteur le plus court, les opérations continuent en reprenant à partir de la première valeur du vecteur le plus court (le vecteur est "recyclé"). Un message d'avertissement vous prévient également lorsque la longueur du vecteur le plus long n'est pas un multiple de la longueur du vecteur le plus court (mais cela n'empêche pas l'opération de se faire).

```r
vec_A <- c(1, 2, 3, 4, 5)
vec_C <- c(1, 2, 3)

vec_A + vec_C # 2 4 6 5 7
# addition : c((1 + 1), (2 + 2), (3 + 3), (4 + 1), (5 + 2))

vec_A - vec_C # 0 0 0 3 3
# soustraction : c((1 - 1), (2 - 2), (3 - 3), (4 - 1), (5 - 2))

vec_A * vec_C  # 1  4  9  4 10
# multiplication : c((1 * 1), (2 * 2), (3 * 3), (4 * 1), (5 * 2))
```

```
vec_A / vec_C # 1.0 1.0 1.0 4.0 2.5
# division : c((1 / 1), (2 / 2), (3 / 3), (4 / 1), (5 / 2))
```

2.5.5 Opérations logiques sur un vecteur

On peut faire des opérations logiques sur des vecteurs (voir les tables de vérité plus haut) :

```
### 1) Est-ce que les éléments de c(1,2,3) sont inclus dans c(1,3,5,7,9) ?
c(1, 2, 3) %in% seq(from = 1, to = 9, by = 2)
# TRUE FALSE  TRUE
# les valeurs 1 et 3 sont bien comprise dans le vecteur c(1, 3, 5, 7, 9),
# mais pas la valeur 2

### 2) Est-ce que les éléments de c(1,3,5,7,9) sont inclus dans c(1,2,3) ?
seq(from = 1, to = 9, by = 2) %in% c(1, 2, 3)
# TRUE  TRUE FALSE FALSE FALSE
# les valeurs 1 et 3 sont bien comprise dans le vecteur c(1, 2, 3),
# mais pas les valeurs 5, 7 et 9

### 3) opération ET entre les éléments de deux vecteurs logiques
c(TRUE, TRUE, FALSE, FALSE) & c(TRUE, FALSE, TRUE, FALSE)
# TRUE FALSE FALSE FALSE
# 1er élément avec le 1er élément     TRUE & TRUE = TRUE
# 2ème élément avec le 2ème élément  TRUE & FALSE = FALSE
# 3ème élément avec le 3ème élément  FALSE & TRUE = FALSE
# 4ème élément avec le 4ème élément FALSE & FALSE = FALSE

### 4) opération OU entre les éléments de deux vecteurs logiques
c(TRUE, TRUE, FALSE, FALSE) | c(TRUE, FALSE, TRUE, FALSE)
# TRUE  TRUE  TRUE FALSE
# 1er élément avec le 1er élément     TRUE | TRUE = TRUE
# 2ème élément avec le 2ème élément  TRUE | FALSE = TRUE
# 3ème élément avec le 3ème élément  FALSE | TRUE = TRUE
# 4ème élément avec le 4ème élément FALSE | FALSE = FALSE
```

2.5.6 Principe de coercition

Commes les valeurs d'un vecteur atomique doivent toutes être du même type, si vous combinez des vecteurs dont les valeurs sont de types différents, R va transformer le résultat dans un seul type de valeur, par **coercition** (*coercion*). Les règles de coercition sont les suivantes :

- combiner un vecteur `character` avec d'autres types (`double`, `integer` ou `logical`) résulte en un vecteur `character` (les valeurs sont toutes transformées en format caractère `"X"`)
- combiner un vecteur `double` (réel) avec un `integer` ou `logical` résulte en un vecteur `double` (les entiers sont transformés en réels, et les valeurs logiques `TRUE` deviennent `1` et `FALSE` deviennent `0`, en format de réels)
- combiner un vecteur `integer` (entier) avec un `logical` résulte en un vecteur `integer` (les valeurs logiques `TRUE` deviennent `1L`, et les valeurs `FALSE` deviennent `0L`, en format d'entiers).

```
vect_dbl <- c(1, pi, 3.54) # un vecteur de réels
vect_int <- c(5L, 4L, -3L, 15L, -8L) # un vecteur d'entiers
vect_char <- c("a", "B", "XYZ", "HELLO") # un vecteur de caractères
vect_logic <- c(TRUE, FALSE, FALSE, F, T, T) # un vecteur logique

## On vérifie ce principe de coercition
vect1 <- c(vect_dbl, vect_int, vect_char, vect_logic)
vect1
typeof(vect1) # "character"

vect2 <- c(vect_dbl, vect_int,  vect_logic)
vect2
typeof(vect2) # "double"

vect3 <- c(vect_int,  vect_logic)
vect3
typeof(vect3) # "integer"
```

2.5.7 Attributs d'un vecteur

Un vecteur a deux principaux attributs :

- sa **classe**, que l'on peut obtenir avec la fonction `class()`
- sa **longueur**, c'est-à-dire le nombre de valeurs qu'il contient, que l'on peut obtenir avec la fonction `length()`

Pour décrire de manière résumé les attributs d'un vecteur, vous pouvez utiliser la fonction `str()` : cette fonction vous indiquera la classe du vecteur, sa longueur et affichera également les premières valeurs. C'est la fonction qui est appliquée pour décrire les objets dans la fenêtre "environnement" de RStudio.

```
## on reprend les vecteurs vus précédemment :
vect_dbl <- c(1, pi, 3.54)
vect_int <- c(5L, 4L, -3L, 15L, -8L)
```

```r
vect_char <- c("a", "B", "XYZ", "HELLO")
vect_logic <- c(TRUE, FALSE, FALSE, F, T, T)

## la fonction class() permet de décrire le type du vecteur
class(vect_dbl) # numeric   # note ici, R indique "numeric" plutôt que "double"
class(vect_int) # integer
class(vect_char) # character
class(vect_logic) # logical

## la fonction length() permet de décrire la longueur du vecteur (c'est à dire
## le nombre d'éléments qu'il contient)
length(vect_dbl) # 3
length(vect_int) # 5
length(vect_char) # 4
length(vect_logic) # 6

## description sommaire des attributs des vecteurs
str(vect_dbl) # num [1:3] 1 3.14 3.54
str(vect_int) # num [1:3] 1 3.14 3.54
# note que la fonction indique "numeric" plutôt que "integer"
str(vect_char) # chr [1:4] "a" "B" "XYZ" "HELLO"
str(vect_logic) # logi [1:6] TRUE FALSE FALSE FALSE TRUE TRUE
```

Il est possible d'ajouter des attributs à un vecteur. Les attributs peuvent être considérés comme des méta-données associées au vecteur.

Un des attributs les plus fréquents est de nommer chaque élément d'un vecteur avec la fonction `names()` : on associe un vecteur de noms (en caractères) aux valeurs du vecteur.

```r
## par exemple, si on crée le vecteur de réels suivant
sex <- c(1, 1, 2, 1, 2, 2)
names(sex) <- c("homme", "homme", "femme", "homme", "femme", "femme")
sex
# homme homme femme homme femme femme     le nom apparaît au dessus des valeurs
#     1     1     2     1     2     2

## mais on peut nommer les valeurs de façons parfaitement arbitraire :
vect_bizarre <- c(1, 2, 3, 4, 5)
names(vect_bizarre) <- c("un", "trois", "douze", "douze", "2")
vect_bizarre
# un trois douze douze     2             le nom apparaît au dessus des valeurs
#  1     2     3     4     5
```

La fonction `attr()` permet d'obtenir ou de définir un attribut spécifique (pour

compléter les méta-données). La fonction `attributes()` indique l'ensemble des attribus associés à un vecteur.

```
## La fonction attr() permet de récupérer des attributs spécifiques
attr(sex, "names")
# [1] "homme" "homme" "femme" "homme" "femme" "femme"

## Cette fonction permet également de définir de nouveux attributs
## Ci-dessous, on définit un nouvel attribut "var_name" auquel on associe
## la valeur "Sexe du participant" pour stocker la méta-donnée indiquant le
## nom complet de la variable.
attr(sex, "var_name") <- c("Sexe du participant")
sex
# homme homme femme homme femme femme
#     1     1     2     1     2     2
# attr(,"var_name")
# [1] "Sexe du participant"

## La fonction attributes() permet de décrire l'ensemble des attributs du vecteur
attributes(sex)
# $names
# [1] "homme" "homme" "femme" "homme" "femme" "femme"
#
# $var_name
# [1] "Sexe du participant"

## On peut récupérer également chaque attribut avec l'opérateur dollar $
attributes(sex)$names
# [1] "homme" "homme" "femme" "homme" "femme" "femme"
attributes(sex)$var_name
# [1] "Sexe du participant"
```

2.5.8 Vecteurs de type `factor`

Les vecteurs de types `factor` sont utiles pour ajouter certaines contraintes propres aux variables qualitatives. Ce sont des vecteurs qui contiennent uniquement des valeurs prédéfinies (connues dès le protocole de l'expérience). Par exemple, en amont de l'expérience, on peut avoir défini que le niveau d'étude se mesurera avec 3 modalités : “lycée”,“bac”, et “université”.

Cette caractérisation sera utile pour utiliser la variable dans des contextes précis, par exemple :

- dans un modèle de régression (où on veut que le logiciel créé automatiquement des indicatrices pour prendre en compte cette variable qualitative de manière adaptée),

- pour représenter graphiquement la variable (par exemple, identifier automatiquement que ces valeurs doivent être décrites avec un diagramme en barres plutôt qu'en box-plot).
- dans des analyes descriptive, cela permettra d'identifier directement si certaines modalités de réponses n'apparaîssent pas la série de valeurs (le décompte sera de 0).

Les vecteurs de type `factor` sont construits par dessus des vecteurs d'entiers, avec 2 attributs :

- un attribut `class`, qui indique "factor" et permet d'identifier ce vecteur en tant que `factor`.
- un attribut `levels`, qui définit les valeurs possibles (définies a priori). Dans notre exemple, ce sont les 3 valeurs "lycée","bac", et "université".

Pour créer un vecteur de type factor, on utilise la fonction `factor()`. Par défaut, R va identifier les 3 valeurs uniques présentes dans le vecteur, puis les ranger de la plus petite à la plus grande (selon la valeur numérique ou l'ordre alphabétique) pour créer l'attribut `levels`. L'attribut levels est un vecteur de caractères.

```
## Exemple avec une variable mesurant le niveau d'étude :
fact_1 <- factor(c("univ", "bac", "bac", "lycée", "univ", "lycée", "bac"))
fact_1
# [1] univ  bac   bac   lycée univ  lycée bac
# Levels: bac lycée univ

## on peut récupérer les 2 attributs de ce 'factor' avec la fonction attributes()
attributes(fact_1)
# $levels
# [1] "bac"   "lycée" "univ"
# $class
# [1] "factor"

## l'attribut 'levels' est un vecteur de caractères, rangé par ordre alphabétique
attributes(fact_1)$levels
# [1] "bac"   "lycée" "univ"

## Un entier est associé à chaque élément de l'attribut 'levels'  :
## "bac" est associé à 1,
## "lycée" est associé à 2,
## "univ" est associé à 3
##
## si on force par coercition à afficher le vecteur au format d'entiers :
as.integer(fact_1)
# [1] 3 1 1 2 3 2 1
```

On aurait sans doute préféré que l'ordre des `levels` commence avec le "lycée" et termine avec "univ". Pour cela il est possible de définir nous même le vecteur de levels avec l'argument `levels` de la fonction `factor` :

```
## On utilise l'argument 'levels' directement dans la fonction 'factor' :
fact_1bis <- factor(c("univ", "bac", "bac", "lycée", "univ", "lycée", "bac"),
                    levels = c("lycée", "bac", "univ"))
fact_1bis
# [1] univ  bac   bac   lycée univ  lycée bac
# Levels: lycée bac univ

# cette fois ci, les levels sont dans l'ordre qui nous convient.

## Un entier est associé à chaque élément de l'attribut 'levels', en suivant
## l'ordre de ses éléments
# "lycée" est associé à 1,
# "bac" est associé à 2,
# "univ" est associé à 3
#
# si on force par coercition à afficher le vecteur au format d'entiers :
as.integer(fact_1bis)
# [1] 3 2 2 1 3 1 2
```

Si la variable qualitative a été codée avec un codage numérique, on peut également appliquer la fonction `factor()`. Prenons l'exemple de la variable "tendance" codée : -1 pour une diminution, 0 pour une stabilité, et 1 pour une augmentation.

```
## Exemple avec une variable mesurant une tendance sous force de codage numérique :
fact_2 <- factor(c(0, 1, 0, -1, 0, 0, -1, 1, -1, 0, 1, 1))
fact_2
#  [1] 0  1  0  -1 0  0  -1 1  -1 0  1  1
# Levels: -1 0 1

str(fact_2)
# Factor w/ 3 levels "-1","0","1": 2 3 2 1 2 2 1 3 1 2 ...
## Après transformation du vecteur en 'factor', les valeurs -1, 0 et 1 vont
## apparaître sous format de caractères "-1", "0" et "1".

## R a identifié automatiquement les 3 valeurs uniques : -1, 0, et 1,
## les a rangé de la plus petite à la plus grande.
attributes(fact_2)
# $levels
# [1] "-1" "0"  "1"
# $class
```

```
# [1] "factor"

# Un entier est associé à chaque élément de l'attribut 'levels'  :
# "-1" est associé à 1,
# "0" est associé à 2,
# "1" est associé à 3
#
# si on force par coercition à afficher le vecteur au format d'entiers :
as.integer(fact_2)
# [1] 2 3 2 1 2 2 1 3 1 2 3 3
```

On peut noter que les entiers sur lesquels les vecteurs `factor` sont construits commencent toujours à 1 et augmente de 1 en 1. Il n'est pas possible de faire autrement (par exemple d'associer une valeur 0 à une des modalités de réponses).

On voit que les vecteurs de type `factor` ne sont pas pratiques pour manipuler de manière flexibles le codage de variables qualitatives (attribuer les codes et les étiquettes de manière flexible). Pour une meilleure gestion des méta-données (des noms de variables, des codages et des étiquettes associées aux modalités de réponses), il est préférable de créer une base de données des méta-données. Nous verrons comment faire au chapitre 3.

Les variables qualitatives peuvent également être définis comme des **facteurs ordonnés** avec la fonction `ordered`. Cela peut être utile pour des variables à utiliser dans le cadre de régression multinomiales par exemple. Le comportement d'un vecteur de type `ordered` est très proche de celui d'un `factor` (le type `ordered` est une petite variation du type `factor`).

```
## La variable de niveau d'étude est une variable qualitative ordinale,
## on peut la définir également en tant que vecteur 'ordered'
fact_1ter <- ordered(c("univ", "bac", "bac", "lycée", "univ", "lycée", "bac"),
                     levels = c("lycée", "bac", "univ"))
fact_1ter
# [1] univ  bac   bac   lycée univ  lycée bac
# Levels: lycée < bac < univ
## on voit que R considère que "univ" est supérieur à "bac", lui même supérieur
## à "lycée"

attributes(fact_1ter)
# $levels
# [1] "lycée" "bac"   "univ"
#
# $class
# [1] "ordered" "factor"
## le vecteur est à la fois de type "factor" et de type "ordered"
```

```
## Un entier est associé à chaque élément de l'attribut 'levels', en suivant
## l'ordre de ses éléments
# "lycée" est associé à 1,
# "bac" est associé à 2,
# "univ" est associé à 3
#
# si on force par coercition à afficher le vecteur au format d'entiers :
as.integer(fact_1ter)
# [1] 3 2 2 1 3 1 2
```

2.5.9 Vecteurs de type `Date` ou `Date-time`

Les vecteurs de type `Date` ou `Date-time` sont construits à partir de valeurs réelles, avec un attribut permettant de les prendre en compte comme des "dates" ou des "dates-heures".

Les **vecteurs de types `Date`** ont un attribut de classe égale à `"Date"`. La valeur réelle sous jacente correspond au nombre de jours depuis le 1er janvier 1970. Le vecteur affiche ses valeurs au format "aaaa-mm-jj" pour indiquer l'année, le mois et le jour, séparés d'un tiret et entre guillemets. On peut saisir des valeurs de dates avec un vecteur de caractères au sein de la fonction `as.Date()`

```
## on saisie un vecteur de 2 dates :
vect_dt <- as.Date(c("1970-01-31", "1971-01-01"))
vect_dt

typeof(vect_dt)
# [1] "double"      les valeurs sous-jacente sont des réels

attributes(vect_dt)
# $class
# [1] "Date"

## si on force le vecteur vect_dt à donner les valeurs au format de réels,
## on retrouve le nombre de jours depuis le 1er janvier 1970.
as.double(vect_dt)
# [1]  30 365
```

Les **vecteurs de types "Date-time"** indiquent une date et une heure. La valeur réelle sous jacente correspond au nombre de secondes depuis le 1er janvier 1970. Ces vecteurs ont deux attributs :

- un attribut de classe égale à `POSIXct` pour *Portable Operating System Inerface - calender time* (ou encore `POSIXlt`, pour *Portable Operating*

System - local time, mais ce format semble moins pratique pour une base de données).
- un attribut de fuseau horaire `tzone` (pour *time zone*). Cet attribut aura simplement un effet sur l'heure affiché, la valeur réelle sous-jacente reste la même.

Le vecteur affiche ses valeurs au format "aaaa-mm-jj hh:mm" pour indiquer l'année, le mois, le jour, l'heure et les minutes, entre guillemets. On peut saisir des valeurs de dates avec un vecteurs de caractères au sein de la fonction `as.POSIXct()`

```
## on saisie un vecteur de 2 dates-heures :
vect_dt_time <- as.POSIXct(c("1970-01-01 01:30", "1971-01-02 00:00"),
                           tz = "UTC") # pour le fuseau du méridien de Greenwich
vect_dt_time # [1] "1970-01-01 01:30:00 UTC" "1971-01-02 00:00:00 UTC"

typeof(vect_dt_time)
# [1] "double"      les valeurs sous-jacentes sont des réels

attributes(vect_dt_time)
# $class
# [1] "POSIXct" "POSIXt"
#
# $tzone
# [1] "UTC"

## si on force le vecteur vect_dt_time à donner les valeurs au format de réels,
## on retrouve le nombre de secondes depuis le 1er janvier 1970 à minuit (00:00).
as.double(vect_dt_time)
# [1]    5400 31622400

# Entre le 1er janvier 1970 à minuit et le 1er janvier 1970 à 1h30, il s'est
# écoulé 60 * 90 secondes = 5400 secondes.

## Si vous souhaitez appliquer le fuseau horaire de Paris à ces valeurs :
attr(vect_dt_time, "tzone") <- "Europe/Paris"
vect_dt_time
# [1] "1970-01-01 02:30:00 CET" "1971-01-02 01:00:00 CET"
## la valeur s'affiche avec 1 heure de décalage (par rapport à Londres)
```

2.5.10 Indiçage d'un vecteur

Pour sélection des sous-ensembles de valeurs au sein d'un vecteur, R utilise un utiliser l'indiçage pour sélectionner les éléments d'un vecteur

fonctions which(), match(), any(), all()

2.5.11 fonctions statistiques pour résumer une série de valeurs

min() max() median() quantile() mean() sd() var() sum() prod()

table() prop.table() unique() sort()

2.6 Les listes `list()`

2.6.1 Création d'une liste

Les vecteurs peuvent être reformatés sous forme de liste

2.6.2 Indiçage d'une liste

2.7 Les matrices `matrix()`

2.7.1 Création d'une matrice

Les vecteurs peuvent être reformatés sous forme de matrice

2.7.2 Indiçage d'une matrice

2.7.3 matrices à plus de 2 dimensions `array()`

2.8 Les bases de données `data.frames()`

2.9 Les objets R peuvent posséder des attributs `attributes()`